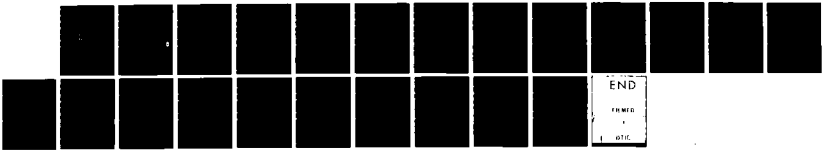
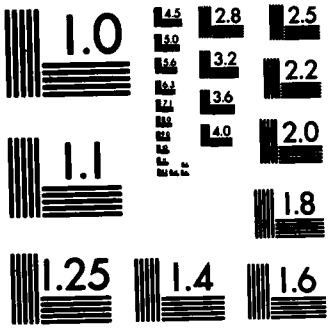
AD-A123 296 THE ARCHITECTURE OF A MULTIPLE REPRESENTATION SYSTEM (DRAFT)(U) STANFORD UNIV CA DEPT OF COMPUTER SCIENCE M R GENESERETH 30 MAY 81 HPP-81-6 N00014-81-K-0004 1/1

UNCLASSIFIED F/G 9/4 NL





Stanford Heuristic Programming Project
Memo HPP-81-6

30 May 1981

The Architecture of a Multiple Representation System (Draft)

by

Michael R. Genesereth

Computer Science Department
School of Humanities and Sciences
Stanford University
Stanford, California 94305

Abstract

MRS is a knowledge representation system intended for use by computer programmers in building models of the world. It provides a single powerful language for stating facts while storing those facts in a variety of different representations. Because of its multiple representations, it is computationally superior to other knowledge representation systems; and, because of its powerful language and inference capability, it is expressively superior to traditional data definition languages

The chief question in building such a system is how the various storage and access routines are tied to the sentences of the formal language. In MRS the system is treated in a domain in its own right. One can write sentences about subroutines and other sentences and allow the system to reason with them, just as it reasons about geology or medicine. Environmental considerations (like running time and storage requirements) and domain and range restrictions can easily be expressed. In practice, MRS uses this "meta-level" information in deciding how to carry out each operation. Thus, one can easily switch representations or inference methods by changing these sentences, and one can implement a variety of different meta-control schemes. Furthermore, the system is completely modifiable in that, by a progression of such changes, it can be converted into any program whatsoever.

MRS is implemented as a practical programming tool in a variety of Lisps. It offers a diverse repertory of commands for asserting and retrieving information, with various inference techniques (e.g. backward and forward chaining) and various search strategies (e.g. depth-first, breadth-first, and best-first search). The initial system includes a vocabulary of concepts and facts about logic, sets, mappings, arithmetic, and procedures. Additional "plug-in" modules are available to handle contexts, default reasoning, and truth maintenance. There is a rudimentary compiler to eliminate needless meta-level processing and a meta-level consistency checker to protect the user from fatal errors in making system modifications.

1. Introduction

In a broad sense, much of computer programming is simulation*. One encodes facts about some aspect of the world and writes programs to answer questions using those facts. This is true whether one is a writing a complex numerical code or an inventory control program or an expert system to do medical diagnosis. In non-numerical programs the selection of a good data structure to encode facts is particularly important and is often more difficult than writing the code that processes it. Over the years, two distinct approaches to representation have arisen, which might be called "theory-building" and "model-building".

The theory-building approach is typified by a growing number of "knowledge representation" systems (e.g. KRL {ref}, KLONE {ref}, Prolog {ref}, UNITS {ref}). Most of these systems provide languages in which the "programmer" encodes facts about the world, including in many cases partial information (e.g. "Arthur is Bertram's father or Allison is his mother.") and quantification (e.g. "All apples are red"). Most offer some general inference capabilities for reasoning about facts expressed within their language (e.g. inheritance in the frame languages). In the vocabulary of formal logic, a set of such sentences is called a "partial theory"; and, if the sentences are true in the world being described, that world is called a "model" of the theory. (See the right hand side of figure 1.) The methodology implicit in most knowledge representation systems might be called "theory-building" because of the emphasis they place on the construction of partial theories.

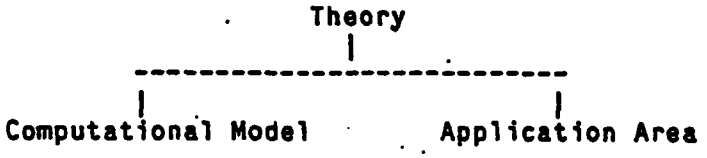


Figure 1 - Theories and Models

In order to simplify their task, most of these systems employ a single data type for encoding facts, e.g. slot-value pairs in frames and links in semantic nets. The problem with this approach is that it doesn't take advantage of the special structure of the knowledge in domains that the system is being used to describe. For example, if one is encoding facts about set membership in a small universe, characteristic bit vectors are a particularly economical representation and allow much more rapid union and intersection computations than slot-value pairs.

{Footnote: Each set in the universe is represented by a vector of bits, each bit corresponding to one member of the universe. The membership of an object in the set is designated by "turning on" the corresponding bit in this vector; its absence is designated by turning the bit off. With this representation, unions and intersections can be computed using hardware or microcoded boolean operations.}

An alternative to this approach is to view data representation as model-building. To describe a world, one constructs a second model of the theory (the left hand side of figure 1), one whose implicit relations mimic those of the world being described. The point of building the model is that it is often easier to answer questions by examining the model than by observing the

world or reasoning with a formal representation of the theory. Thus, for example, an architect builds 3-D models to help himself and his clients envision spatial relationships. Navigators use charts to fix their positions. Engineers once used slide rules to do multiplication, slide rules being computational models of the theory of logarithms and multiplication.

Model-building is the traditional approach in computer programming, except that the models are built within the computer rather than in the external world. The programmer selects or designs a data structure to encode the essential information about his world and writes code to process it. For example, he might decide on a tree structure to represent an organizational hierarchy or a bit vector to represent a set, as suggested above.

Because of the difficulty of choosing good data structures in complex applications, the programming language community has for some time recognized the desirability of separating the choice of representation in a program from the structure of the code that processes it. With this separation different representations can be tried and modifications can be made without rewriting the code. This realization led to the introduction of various "record packages" (e.g. in EL-1 {ref}, Pascal {ref}, Interlisp {ref}, or Lisp Machine Lisp {ref}). For example, ... In addition to separating data representation from code, this approach has the advantage of making data access more mnenomic.

The major shortcoming of this approach is that it doesn't allow the encoding of quantification or partial information. Furthermore, none of the existing systems provides any sort of inference, except for specialized procedures written by the programmer. While there may be little need for general inference in finshied programs, such a capability is highly useful during development.

As computers are applied to increasingly complex tasks (particularly in AI), the deficiencies of these two approaches in isolation are becoming more significant. Fortunately, they can be merged in a way that eliminates their shortcomings. The key idea is provide the programmer with a theory-building language, like predicate calculus or one of the other AI languages, while storing the facts stated in this language in a variety of different ways. Some facts may be stored in the language verbatim; others may be stored by modifying an appropriate computer model. Correspondingly, some facts are looked up in the theory language; others, in the model. Alternatively, they may be deduced using general inference procedures on the theory language or specialized procedures in the model. Figure 2 presents the situation graphically.

{Footnote: The possibility of multiple languages as well as multiple representations is discussed further in section ?.}

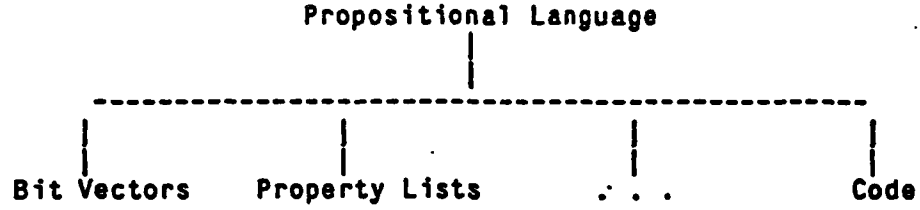


Figure 2 - A Multiple Representation System

This paper describes the architecture of a particular multiple representation called MRS. Section 2 describes MRS's language and theory-building facilities; section 3 discusses model-building; and section 4 shows how MRS's architecture makes it completely modifiable. Section 5 summarizes the state of implementation and indicates some directions for future work.

2. Theory-Building in MRS

MRS is a multiple representation system implemented in Lisp. It is a practical programming tool intended for use by AI researchers in building expert systems. It offers a diverse repertory of commands for asserting and retrieving information, with various inference techniques (e.g. backward and forward chaining) and various search strategies (e.g. depth-first, breadth-first, and best-first search). The initial system includes a vocabulary of concepts and facts about logic, sets, mappings, arithmetic, and procedures. Additional "plug-in" modules are available to handle contexts, default reasoning, and truth maintenance. This section describes only those features of the system relevant to the succeeding discussion. For further details, the reader should consult the MRS manual {ref}.

2.1 Syntax

MRS's language is a prefix version of the language of predicate calulus. The syntax is straightforward and should be apparent from some examples. Each symbol in an assertion is either a variable or a constant and can stand for an object, a function, an action, a relation, a relation on relations, a relation on actions, etc. Each complete lists makes up a formula. The examples in figure 3 show how the various types of formulas in first order predicate calculus are expressed. Atomic formulas are represented by grouping symbols together with a relation symbol of the appropriate type. Non-atomic formulas are expressed by relating several formulas with a logical symbol.

Bertram and Arthur are neighbors.
(neighbor bertram arthur)

Carleton is not Bertram's neighbor.
(not (neighbor carleton bertram))

Either Carleton is Bertram's father or Beatrice is his mother.
(or (father-of carleton bertram) (mother-of carleton beatrice))

All apples are red.
(all x (if (mem x apples) (color-of x red)))

```
Some apple is green.
(some x (and (mem x apple) (color-of x green)))

Everybody has someone he loves.
(all x (some y (loves x y)))

There is someone whom everybody loves.
(some y (all x (loves x y)))
```

Figure 3 - First Order Predicate Calculus

Higher order formulas are formed either by using functional or relational variables or by applying higher order relations to functional or relational constants. The induction axiom shown in figure 4 is a simple example.

```
Whenever 0 is in a set and n's membership in the set implies the
membership of n+1, then every natural number is in the set.

(all s (if (and (s 0) (all x (if (s x) (s (+ x 1)))))
     (all x (s x))))
```

Figure 4 - An Example of a Second Order Formula

Two useful syntactic features are illustrated in figure 5. The first is the use of the prefix characters $ and ? to denote universal variables and existential variables without universals quantified to their left. Second, a function in MRS can be used in two ways, either with its arguments in places reserved for individuals or in assertions with its value as the last argument.

```
Every neighbor of Bertram is a neighbor of Beatrice.
(all x (if (if (neighbor x Bertram) (neighbor x Beatrice)))
(if (neighbor $x Bertram) (neighbor $x Beatrice))

Some apple is green.
(some x (and (mem x apples) (color-of x green)))
(and (mem ?x apples) (color-of ?x green))

Beatrice is Carleton's mother.
(mother-of Carleton Beatrice)
(= (mother-of Carleton) Beatrice)
```

Figure 5 - Some abbreviations

In MRS knowledge representation is an application domain in its own right, just like geology or medicine. The system contains knowledge about its own structure and behavior and some popular variations. This information is encoded within the MRS formalism itself, and as a result the system can apply to it the same deductive routines that it uses in reasoning about external domains.

As described above, symbols in MRS are used to designate objects in the applications domain. In order to state facts about a symbol itself, one needs a separate symbol to represent it. In MRS these "metasymbols" are by convention named by prefixing the symbol represented with an ↑. Consider, for example, the symbol bachelors that represents the set of all bachelors and the symbol ↑bachelors

that stands for that symbol. Clearly, it is appropriate to add the assertion (mem bachelors set), and certainly it is the case that (not (mem ↑bachelors set)). If bachelors and ↑bachelors were coalesced, a contradiction would result. Similarly, one might have an assertion about the size of a person named John, e.g. (size John large), and a contradictory assertion about the size of the symbol representing John, e.g. (size ↑John small). In MRS a symbol and its metasymbol are connected by the denoted-by relation, e.g. (denoted-by John ↑John).

Terms, like symbols, designate objects in the application domain; and, as ith symbols, a separate symbol is necessary whenever one wants to state a fact about the term itself. The ↑ convention can be used here as well. For example, the term (color-of clyde) designates a point on the visual spectrum, whereas the symbol ↑(color-of clyde) designates an atom in MRS.

Since propositions don't refer to objects in the application domain, they can be used to designate themselves when used as an argument in another proposition. Figure 6 shows some examples of this in representing dependency and theory information.

```
(dependency (resistance r1 100) (voltage v1 5))
(MyTheory (president US Kennedy) 1961-1963)
```

Figure 6 - Some meta-assertions

In order to talk about propositions, a few special vocabulary items are defined. (Rel p) designates the symbol in the relational position of the proposition; (arg p) designates the symbol in the argument position; and (val p) designates the symbol in the value position. (Prop ↑r ↑a1 . . . ↑an) designates the proposition symbol made up of the indicated symbols, i.e. (r a1 . . . an).

2.2 Subroutines

Propositions can be stored and removed from MRS's data base using the subroutines stash and unstash. The lookup subroutine checks whether a proposition is in the data base and returns ((t . t)) if succesful and nil otherwise. If the proposition contains existential variables, lookup returns a binding list for those variables. Lookups (the plural) returns a list of all bindings for which there is a proposition in the data base. Consider the following examples.

```
(stash '(neighbor Arthur Bertram))
(neighbor Arthur Bertram)

(stash '(neighbor Arthur Billings))
(neighbor Arthur Billings)

(lookup '(neighbor Arthur Billings))
((t . t))

(lookup '(neighbor Arthur ?x))
      or
(lookup '(exist ?x (neighbor Arthur ?x)))
((?x . Bertram) (t . t))

(lookups '(neighbor Arthur ?x))
(((?x . Bertram) (t . t))  ((?x . Billings) (t . t)))
```

As a convenience for its users, MRS offers a limited amount of automatic inference. Whenever a fact is asserted, the system may infer other facts and assert them as well. Whenever a user asks MRS whether a fact is true, the system may be able to deduce it even though the fact is not explicitly stored. The subroutines assert, unassert, truep, and trueps all perform inference in processing their arguments. Consider the following examples.

```
(assert '(elephant clyde))
(elephant clyde)

(assert '(if (elephant $x) (color-of $x grey)))
(if (elephant $x) (color-of $x grey))

(truep '(color-of clyde ?y))
((?y . grey) (t . t))
```

The system's default inference method for retrieving information is depth-first backward chaining. However, a number of other inference methods are defined as well, and the user can instruct MRS to use any one of these or one of his own, as described in section 4.

3. Building Models in MRS

3.1 Theories and Models

In formal logic, a theory is defined to be a set of sentences closed under logical implication. In what follows, this definition will be broadened to include any set of sentences closed under inference by an arbitrary inference procedure. If the procedure is complete (like resolution), the two notions are equivalent, but this is not the case with the inference methods in most current representation systems.

The correctness of a theory depends upon the interpretation one assigns to its sentences. In Tarskian semantics {ref} an interpretation consists of a set of objects and a mapping from the symbols of the theory's language to objects in this set, functions on those objects, and relations among them. An interpretation is a model of a theory if the sentences are all true under the interpretation. Of course, there may be more than one model for a given theory. For example, the integers under addition and the rational numbers under multiplication are both models of the theory of groups.

In the theory-building approach to knowledge representation, the task is to encode enough facts and a sufficiently powerful inference procedure that the resulting theory covers the space of possible questions. The inference procedure is usually written by the system's builder, and the facts are usually entered by either the system builder or its user. For example, the builder of a medical diagnosis system encodes inference rules and general facts about medicine, and the user inputs facts about a specific patient and asks for the system's conclusions.

In the model-building approach, the task is to build a second model of the intended theory. This might be another physical model (e.g. an architect's sketch), or it might be a

set of data structures within a computer (e.g. a bit array). Most knowledge representation systems deal with computer models; but, with adequate sensors and effectors, they could produce and utilize external physical models equally well.

One of the limitations of the model-building approach is that there are sentences that cannot be represented in a model, e.g. a universally quantified formula. Such facts may just happen to be true of all the objects currently in the model. Alternatively, if the model is a good one, such facts may be implicit in the representation. For example, if a linear list is chosen as a representation of a linear order, it's imposssible to state a fact that violates the trichotomy rule. However, neither of these is an explicit representation for this fact.

In addition, there is a difficulty in dealing with partial information. One might, for example, know that A is greater than C and B is greater than C in a linear order, but unless one can determine the order of A and B, it's impossible to represent both of these facts in a linear list representation.

{Footnote: The requirement that certain information be available before a fact can be asserted in a model is remisicent of the obligatory character of some slots in Minsky's frame theory {ref}. An important difference is that, in Minsky's formulation, this requirement is an explicit feature of each slot, whereas in model-building it is implicit in the representation.}

In a multiple representation system, quantified formulas can be stored explicitly in the theory language. Furthermore, if the fact is not implicit in the model, the system can automatically enforce it by not allowing any modification to the model that violates it. Partial information can also be encoded in the theory language, using logical operators, until all ambiguity is resolved. Of course, so long as there is a need to represent facts in this language, it's necessary to provide an inference method for reasoning with these sentences, and it's necessary to check both the theory and the model in accessing the data base. This is the major reason for the existence of two distinct sets of commands in MRS: stash, unstash, and lookup for accessing models and assert, unassert, and truep for doing inference beforehand.

The primary question in implementing such a system is how the the representations are chosen. One might imagine a system that could automatically select an appropriate representation. Some exploratory steps in this direction have already been taken by Barstow {ref}, Low {ref}, and Rovner {ref}; and there is an effort underway at Stanford {ref} to study the question in greater generality. However, so far, the results are limited. The alternative is to have the programmer specify a desired representation for each sentence or set of sentences. Then, the system need only look up the representation subroutines whenever it is processing a sentence. This latter alternative is really a subset of the former because, even if the system designs its own data structures, it will need to record its decisions for later access and modifications. In either case, there is a need to represent information about how to process a sentence and a procedure for accessing this information. A specific representation and procedure will be proposed below after a brief but essential digression on object-oriented programming.

3.2 Object-Oriented Programming and Meta-Level Reasoning

Computer models include not only data structures but also the operations one can apply to them. In complex domains, different implementations may be necessary to carry out an operation for different inputs. For example, the sign of a complex number might be computed in a different way from the sign of multivariate polynomial. A common approach in this situation is to define as many different subroutines as necessary and write a dispatching routine that examines the inputs and calls the appropriate one. This approach of determining the ubroutine to be used on the basis of the object it is to be applied to is often termed "object-oriented programming".

Both the programming language and artificial intelligence communities have developed programming systems that facilitate this style of programming. In Simula, subroutines are associated with classes of objects organized into a subclass hierarchy; and the system uses property inheritance (forward chaining over specified relations, usually membership and subset) over this hierarchy to retrieve appropriate subroutines. In Planner, Conniver, QA-3, and QA-4, the programmer defines subroutines for adding, deleting, and retrieving data base assertions; and the system uses "pattern-directed invocation" (a subset of universal instantiation and existential generalization) to associate these subroutines with specific data base assertions. In the frame-based systems, the user is able to attach subroutines to the "slots" of each frame, and these procedures are inherited from frame to frame much as in Simula.

One shortcoming of these systems is the limited amount of inference they employ in subroutine lookup. The property inheritance method used by Simula and the frame systems works well when one is computing a function of a single argument, but it encounters difficulty when more than one object is involved, e.g. in adding two numbers of a different type, say a complex number C and a rational number R. Should the subroutine be inherited from C or from R or perhaps from +? What if there is a conflict? Some systems, like FRL, use inheritance on the argument of the operation (in this case C or R); others, like RLL, use the operation itself (here +). Each of the systems can simulate the other but only at some loss in naturalness. The ideal would be to allow any combination of components to determine the representation.

The Planner approach is slightly more general in that it allows all of the objects to be taken into account. However, its pattern matching inference is less effective and fails when one wants to use partial information. For example, the programmer may want to specify that table lookup be used in computing any binary relation of the objects S and T and not just membership or subset.

A further limitation is that there is no easy way of getting any of these systems to take environmental considerations into account in selecting subroutines. Yet, one might want to carry out an operation differently depending on the time of day or the identity of the operating system or the amount of storage left. In fact, there isn't even any way of stating such factors, e.g. in the definition of a Simula class.

An approach that eliminates these shortcomings is to treat the system as an application area in its own right, like hydrodynamics or truck scheduling or medicine. In this way, operations and subroutines can be talked about as objects, as well as the time of day, the operating system, and the amount of storage left, and one can write statements relating operations to subroutines and allow the system to reason about the choice of subroutine just as it reasons about external domains.

In MRS the association between an operation F and a subroutine S that implements it for inputs a1, . . ., an is written as an assertion of the form (MyTo F a1 . . . an S). In making such assertions one can take advantage of the full expressive power of MRS's language. For example, one can state that a subroutine S computes the function F for all inputs via the first sentence below. The second sentence states that S works in the more specific case where the second input is B. Using logical operators one can impose partial constraints as well, as in the third sentence. The fourth sentence illustrates the incorporation of environmental factors.

```
(MyTo F $x $y S)

(MyTo F $x B S)

(if (R $x) (MyTo F $x $y S))

(if (and (R $x) (< (cost (action S $x $y)) 100)) (MyTo F $x $y S))
```

{Footnote: The MyTo relation is a special case of a more general ation between "goals" and "plans". Specifically, each abstract ope ion is defined by its goal, and each goal has an associated set of steps, plan, that achieves it. This relationship between goals and plans nt be recorded via an assertion of the form (MyToAchieve goal plan). For mple, the assertion (MyToAchieve (= (lhs $x) $y) (action rplaca $x $y)) s hat in order to change the left hand side of a dotted pair one should us aca. The MyTo relation defined above is a special case of this MyToAchieve relation where the plan is a single subroutine that has the same arguments as the abstract operation it carries out.}

In practice, MRS uses such assertions in deciding how to carry out each operation. The exact mechanism for this should be clear after a look at how MRS works. Figure 8 presents the initial LISP definitions of a particular operation F. F calls the subroutine kb with its corresponding MyTo relation and its argument. Kb first uses backward chaining to decide how to carry out the operation and extracts it from the binding list using subvar. It then applies the result to the specified argument.

```
(defun f (x) (kb 'MyToF p))

(defun kb (f x)
       (funcall (subvar '? (bs-truep '(MyTo f x ?)) x))
```

Figure 8 - Definitions of MRS's user level subroutines

As a very simple example, consider how MRS handles a request like (F A). A trace of the relevant function calls is shown in figure 9. F calls kb with arguments MyToF and A. Kb uses bs-truep to determine how to look carry out F and finds S. It then applies S to A to get the answer.

```
(enter F A)
    (enter kb MyToF A)
         (enter bs-truep (MyToF A ?))
         . . .
         (exit bs-truep S)
         (enter S A)
         . . .
         (exit S B)
```

```
    (exit kb B)
(exit F A)
```

Figure 9 - A trace of function calls in carrying out an operation

An exciting consequence of this architecture is that it allows the system to think before it acts. Backward chaining is quite limited, but one can imagine more sophisticated reasoning.

Suppose, for example, the user had made the assertions (MyTo f $x g) and (MyTo g $x f). With kb defined as shown above, the system would go into an infinite loop, first trying to compute f using g, then trying to compute g using f. One could forestall this problem in many common cases by using a more complicated retrieval that first checks whether the indicated method will halt. See figure 15. If MRS can prove that the indicated method does not halt, it can notify the user rather than embarking on the endless computation.

```
(defun kb (f x)
  (setq f (subvar '? (bs-truep (list MyTo f x '?)))
  (cond ((bs-truep '(non-halting f x)) (print 'Nonhalting))
        (t (funcall f x))))
```

Figure 15 - Meta-reasoning to prevent an infinite loop

Another example of more substantial meta-level reasoning is the choice of one search method over another. Suppose, for example, that one had encoded an ancestor hierarchy and needed to determine whether or not beatrice is an ancestor of clyde. To do this, the system must decide whether to search beatrice's descendants or clyde's ancestors. In making this decision, it could reason that the branching factor in the upward direction is less than the branching factor downward and, therefore, it's quicker to search upwards.

Ultimately, it might be possible for the system itself to write the code to carry out an operation. Recent research has yielded a variety of planning and program synthesis techniques that should be of value in this regard. Each operation in the domain can be characterized by the goal it is intended to achieve. For example,...

```
(if (effect $c (R A B)) (instance $c F))
```

Each available subroutine can be characterized by its side effects and the relationship between its inputs and outputs. For example, ...

```
(if (and (instance $c S) (precondition $c (Q C D)))
    (effect $c (R A B)))

(if (instance $d T) (effect $d (Q C D)))
```

Using this information and a store of planning expertise, a system should be able to synthesize a program to achieve the goal of a given operation, in this case using T to achieve the prerequisite for S, which achieves the desired effect.

These examples are not intended to suggest that these problems are easy or that MRS makes them any easier. Indeed, researchers have spent many years making gradual progress toward their solution. The point here is that MRS provides a framework in which such techniques

can be readily incorporated as they become available.

3.3 Compiling for Efficiency

The disadvantage of a meta-level architecture is the additional cost incurred in looking up the subroutines associated with an operation. However, in a production system, the appropriate subroutine can often be determined in advance (as the result of type declarations, or the corresponding assertions in the representation language), and then this cost can be compiled away. In point of fact, programmers using untyped systems usually do this compilation themselves in writing their subroutines.

In order to regain some of this lost efficiency, MRS has a rudimentary source-to-source translator that eliminates meta-level processing where possible. The user specifies which facts he wants the translator to use by declaring them to be "frozen". The translator then uses these facts to optimize the programs it is called on. At this writing, the translator uses three optimization techniques: constant folding to incorporate the frozen information, symbolic evaluation to propagate it, and peephole optimization to eliminate any obvious inefficiencies uncovered by the other two techniques.

Consider, for example, the problem of optimizing the code (denom (plus A B)) propositions and definitions shown below.

```
(defun rattimes (x y)
       (cons (* (car x) (car y)) (* (cdr x) (cdr y))))

(if (and (rational $x) (rational $y))
    (MyTo times $x $y rattimes))
(if (and (rational $x) (rational $y))
    (rational (output (action rattimes $x $y))))
(if (rational $x) (MyTo denom $x cdr))

(rational A)
(rational B)
```

Using constant folding, the first MyTo assertion, and the type declarations on A and B, the call to times is replaced by rattimes. A bit of symbolic evaluation shows that the output is also a rational number, and so the call to denom can be replaced by cdr. The translator then open codes the definition of rattimes and finds that it can apply peephole optimization to get the final form shown.

```
(denom (times A B))

(denom (rattimes A B))

(cdr (rattimes A B))

(cdr (cons (* (car A) (car B)) (* (cdr A) (cdr B))))

(* (cdr A) (cdr B))
```

In order to avoid excessive growth in the size of code, the translator has an arbitrary limit on the amount of open coding it will permit. The translator also retains information connecting the optimized code to the assertions on which it depends and retranslates the code whenever the user unasserts any of the frozen assertions, using the mechanism described in

section 4.3

4. Modifiability

The importance of object-oriented programming in the design of a multiple representation system is clearest when one views the system from the meta-level. As with hydrodynamics and medicine, there is a theory of the system, which describes its operation and structure; and the sentences in this theory can be encoded in the system itself. For example, there might be a proposition stating that, whenever the system contains facts of the form "x is the brother of y" and "y is the mother of z", it also contains the fact "x is the uncle of z".

The connection to object-oriented programming comes when one considers the operations possible in the system, e.g. asserting, retrieving, or deleting a proposition. The way these operations are carried out may depend upon the class of propositions to which the argument belongs. The MyTo relation introduced in the last section can be used to associate a particular subroutine with each class of propositions and kb can then be used to retrieve this information and carry out the operation.

4.1 Simple Modifiability

The advantage of this approach is that it is easy for the system's user to modify the way a class of propositions is handled. He simply makes a different MyTo assertion. Of course, this approach can be used for all of the system's operations, not just data representation. For example, one could assert different MyTo assertion for the assert operation to specify a different inference algorithm; and one could assert various "MyTo Edit" properties to associate different editing subroutines with different objects or propositions. The following examples indicate how some of the more common knowledge representation features can be introduced in MRS using this approach.

Example - Changing the Representation

While the propositional representation is adequate for representing all types of information, property lists are a good example of a specialized data structure that is especially suited to storing the values of unary functions. For example, the assertion (color-of Clyde grey) can be represented by a color-of property on Clyde's property list. The commands in figure 11 show how property lists can be utilized in MRS. In particular, they name the LISP subroutines for accessing and modifying property lists. Of course, its is necessary to assert that a relation is a unary function (either directly or indirectly) in order for this information to be found.

```
(assert '(if (unaryfun (rel $p)) (MyTo stash $p putp)))
(assert '(if (unaryfun (rel $p)) (MyTo unstash $p remp)))
(assert '(if (unaryfun (rel $p)) (MyTo lookup $p chkp)))

(defun putp (p) (putprop (cadr p) (caddr p) (car p)))
(defun remp (p) (remprop (cadr p) (car p)))
(defun chkp (p) (eq (caddr p) (get (cadr p) (car p))))
```

Figure 11 - Storing unary functions on property lists

Example - Changing the Inference Algorithm

Suppose one wanted to convert MRS into a data base system without inference. To do this, the only change that is necessary is to change the system's MyTo truep property to direct lookup, as shown below. Thereafter, whenever truep is called, kb will find lookup, and no further deduction will be performed.

```
(unassert '(MyTo truep $p bs-truep))
(assert '(MyTo truep $p lookup))
```

Figure 10 - Switching from backward chaining to direct lookup

Example - Procedural Attachment

Procedural attachment is the association of a procedure with a "slot" in a "frame" that is executed whenever a value is added to, needed for, or removed from that slot. In MRS procedural attachment is effected by appropriate MyTo Assert, MyTo Unassert, and MyTo Truep properties on the relation corresponding to the "slot". For example, suppose one wanted to build a kinship data base in which only mother-of and spouse-of links were stored, leaving father-of to be computed. In MRS this could be implemented by placing the appropriate properties on father-of as shown in figure 2. When truep is called with (father-of clyde ?y) as argument, kb finds the truep-father procedure and applies it.

```
(assert '(MyTo truep (father $x $y) chkfather))

(defun chkfather (p)
  (truep '(and (mother ,(cadr p) ?y) (spouse ?y ,(caddr p)))))
```

Figure 12 - Computing the father relation

Of course, this could have been done more generally as shown in figure 13. Father is asserted to be a member of the set of ComposedFunctions and is defined as the composition of spouse and mother. All composed functions have truep-composed as their MyToTruep property. Therefore, when truep is called on (father clyde ?y), kb finds truep-composed, which computes the answer as before. The difference here is that, to implement a new composed function, one need only assert that it is a member of ComposedFunctions and indicate the composing functions.

```
(assert '(mem father-of ComposedFunctions))
(assert '(composition spouse mother father))
(assert '(if (mem (rel $p) ComposedFunctions)) (MyTo truep $p chkcomp))

(defun chkcomp (p)
  (prog (dum)
    (setq dum (truep '(composition ?f ?g ,(car p))))
    (return (truep (list 'and
                         (list (subvar '?f dum) (cadr p) ?z)
                         (list (subvar '?g dum) ?z (caddr p)))))))
```

Figure 13 - Implementation of composed functions

Example - Truth Maintenance

Another example is the partial implementation of truth maintenance. Suppose, for example, that dependency information is stored as suggested in figure 6, i.e. as dependency propositions connecting each assertion to the

propositions it depends on. (For simplicity, the possibility of multiple justifications is ignored in this example.) One can define a truth maintenance subroutine devil that removes assertions from the data base whenever their support is removed. The code for devil and the appropriate meta-assertion are shown in figure 14. As a consequence of this assertion, every time Unassert is called, devil will be used and the dependencies will be removed.

```
(assert '(MyTo unassert $x devil))

(defun devil (p)
  (unstash p)
  (mapc '(lambda (l) (unassert (subvar '? l)))
        (lookups '(dependency ,p ?))))
```

Figure 14 - Partial implementation of Truth Maintenance

4.2 Complete Modifiability

The modification of "MyTo" properties is one way that a system can be made modifiable. An intriguing consequence of a multiple representation architecture is that it makes the system modifiable in a much broader sense as well. The system is a model of its own theory, just as the human body is a model of the "theory" of physiology. However, unlike physiology and the human body, the system can observe and modify itself directly. As a result, there is no need to create a second model as suggested in figure 1. The system can examine itself to answer questions and can modify itself in storing assertions. For example, the meta-proposition (Inferrable (R A B)) can be evaluated by trying to infer (R A B). The proposition (Indb (R A B)) can be stored by storing (R A B) using whatever representation is appropriate for it.

{Footnote: If the system had appropriate sensors, it could use this technique to answer questions about the external world as well. One reason for not doing this, even if it had such sensors, is that the test may be expensive. For the same reason, one might not want to check whether a fact is inferrable by trying to infer it. However, the decision about whether to use meta-inference or direct observation (in this case, inference) is beyond the scope of this paper. The point is that it's possible to have it either way.}

In a similar fashion, one can write propositions describing the code of the system. If the system contains the appropriate "MyTo" subroutines, these propositions can be evaluated by examining the code directly; and they can be stored by modifying the code, i.e. it's possible to *reprogram the system merely by making assertions in* its own language. With enough assertions, the system can be converted into any other program. In short, it's completely modifiable.

The mechanism for this is straightforward. One can easily describe Lisp code as sets of assertions. Each s-expression is represented as an individual "action" with an operator, inputs, outputs, and controlflow. For example, the subroutine demon-stash defined below can be described by the assertions that follow. This translation can be carried out by an automatic procedure listed as the "MyTo Lookup" property of code-for sentences.

{Footnote: Note that the existence of this automatic translator means

that one can write procedures in Lisp but still have the system reason about them as if they had been defined in assertions.}

```
(defun demon-stash (p)
       (stash p)
       (rundemons p))

(code-for demon-stash proc23)
(part s1 proc23)
(opr s1 stash)
(= (input 1 proc23) (input 1 s1))
(part r1 proc23)
(opr r1 rundemons)
(= (input 1 proc23) (input 1 r1))
(before s1 r1)
```

In order to allow the user to change the definition of subroutines, there is a simple reverse translator listed as the MyTo Stash property of code-for sentences. Suppose, for example, that one wanted to reverse the order of the steps in demon-stash. One could do this by defining a new procedure proc24 with steps s2 and r2 such that (before r2 s2). Then one asserts (code-for demon-stash proc24) and lets the automatic translator do the rest. Alternatively, one could simply remoev the assertion (before s1 r1) and assert its opposite (before r1 s1). If appropriate dependency links were kept between this order assertion and the code-for sentence about demon-stash, this would cause a retranslation to take place, thus updating the Lisp definition.

Of course, some programming languages, like Lisp, are also completely modifiable. The difference is that Lisp is purely procedural -- there's no way of making "declarative" statements about the external world or the language itself.

4.3 Meta-Level Consistency Checking and Modification Discipline

One of the optional features of MRS is meta-level consistency checking. Whenever the user asserts or deletes a fact, the system checks whether the modification will violate any meta-level assertions and, if so, warns the user. For example, suppose the user had asserted the uniqueness property for functions as shown below. Then, if there were sentences in the data base of the form (color block1 blue) and (function color) and the user tried to assert (color block1 red), the system would catch the contradiction and offer to unassert one of the above sentences.

```
(if (and (indb ($f $x $y))
         (inferrable (function $f))
         (not (= $y $z)))
    (not (indb ($f $x $z))))
```

The same mechanism is used to ensure consistency of the system's theory of itself and thereby protects the user against unintentional omissions. One common problem is for the user to assert a special MyTo Stash property without modifying the MyTo Unstash or MyTo Lookup properties accordingly. This problem can be avoided by relating the various subroutines to each other with assertions like those below. Then, when the user changes one of these properties, the system will detect the inconsistency and offer to change the others or abandon the request.

```
(iff (repr $p plist) (indb (MyTo stash $p putp)))
(iff (repr $p plist) (indb (MyTo unstash $p remp)))
(iff (repr $p plist) (indb (MyTo lookup $p chkp)))
```

Another common mistake is for the user to delete the general MyTo Stash property, making it impossible to assert a new one. One way to forestall this possibility is to add an assertion stating that there must always be a MyTo Stash property, as follows.

```
(all p (exist g (Inferrable (MyTo stash p g))))
```

The nemesis of a modifiable system is its fragility. When a user makes a mistake, he can easily break the system. Consistency checking and meta-level assertions like these are essential in making the system a useable tool. Of course, these properties can be deleted like any others, thus defeating the protection mechanism, unless they themselves are protected by similar assertions.

5. Conclusion

5.1 State of Implementation

MRS is implemented as a core system, together with a set of "plug-in" modules. The core system is very small and contains the minimal code necessary for its meta-level architecture. This includes the definitions of kb and the user-level subroutines, inference via pattern matching and backward chaining, and various low-level subroutines for MRS's propositional representation. The "plug-in" modules modify the system to augment its capabilities. Currently, there are modules that set up other representations (e.g. property lists), other inference procedures (e.g. forward chaining via demons, rewrite rules, resolution), and other search procedures (e.g. breadth-first and best-first). User utilities include a compiler, a "unit" editor, and an error-checking "front-end".

The system is implemented in a variety of Lisps, including Maclisp and Interlisp on the Dec-20, Interlisp on the Xerox Dolphin, and Franz on the Vax. There are also efforts underway to make it available on IBM equipment in Lisp370. The system is being used in a variety of projects at Stanford and elsewhere (e.g. computer diagnosis and intelligent operating system interface) and is taught in the first graduate level AI course at Stanford.

5.2 Future Work

One of the reasons for building MRS was to have an adequate basis from which to do further research in representation. The following paragraphs summarize a few directions for future work.

Section 3.2 mentioned the possibility of more extensive meta-level reasoning in subroutine selection. David Smith {ref} is building on the pioneering work of McCarthy and Hayes {ref}, Davis {ref}, Wilensky {ref}, and others in developing powerful meta-level heuristics for controlling search, recognizing unsolvability, monitoring and analyzing plan execution to learn from experience, etc.

Steve Tappel {ref} is interested in problem reformulation; and, as part of that interest, he is trying to automate the selection of data representation. This involves reasoning about each representation to determine whether it preserves enough information to solve the desired class of problems; it requires strategies for interfacing different

representations; and it requires strategies for comparing possibilities to select the one most suitable. Tappel's very general approach to this problem should become crucially important as VLSI research opens the possibility of heterogeneous systems of processors and thereby engenders a need for new compiling techniques.

The use of specialized representations gives MRS an economy and efficiency not possible with a uniform representation. The economy can be expressed in terms of the space saved due to the use of relations implicit in the specialized epresentation(as the length of a list reflects the degree of teh polynomial it represents). The efficiency in doing deductions is attributable to specialized algorithms. An interesting possibility suggested by this economy and efficiency is for the program to use these criteria in evaluating plausible hypotheses about a new domain. In the face of incomplete or contradictory data, the program might favor the theory with a more economical representation. Clearly, there is some evidence for this sort of behavior in human cognition. Consider, for example, Mendeleev's discovery of the periodic table of the elements. He was convinced of the correctness of the format in spite of contradictory data, for reasons that can only be identified as simplicity, or economy. The key point here is that the existence of multiple representations may affect the functionality of a program as well as its efficiency. This possibility is discussed in greater detail in {ref}, but so far no attempt has been made to incorporate the economy heuristic into a theory formation program.

Finally, as in any discussion of meta-level issues, there is the inevitable question of the meta-meta-level and beyond. As explained above, it's possible to encode propositions about propositions in MRS, and changing these propositions can affect the system's behavior. Naturally, one can encode propositions about propositions about propositions, and the system can reason with these just as well. The interesting question is whether this reasoning can affect the system and, if so, how. As described earlier, each base-level operation takes meta-level propositions into accoutn by calling kb and letting it reason about how to carry out the operation. One way in which meta-meta-level information could be used would be to have KB call itself or yet another kb. The problem with this approach is that the former possibility results in an infinite recursion and the latter implies that the number of levels is limited. While a limited number of levels may be OK for most applications, an alternative that avoids the infinite loop and the level limit is to interleave meta-level reasoning with a default base-level procedure, as suggested in the following definition. Here, bor is a "breadth-first" or, which time shares the execution of its arguments until one returns a non-null value.

```
(defun newkb (g x)
       (bor (kb g x) (newkb 'kb (list g x))))
```

Although there are a number of problems with this approach, it does allow the system to reason at an arbitrary number of levels. Further study is needed to determine whether it can be used to any advantage.

{Footnote: Because MRS uses itself as a model of its own theory, the system conatins an infinite numbor of meta-level propositions. Consider, for example, the propositions

```
(R A B)
(InDB (R A B))
(InDB (InDB (R A B)))
   . . .
```

MRS stores and retrieves propositions with InDB as relation by storing and retrieveing the argument. Thus, whenever (R A B) is in the data base, so are the other propositions shown. Of course, not all InDB propositions can be stored this way; consider, for example, (exist p (InDB p)), which can't be stored without knowing the identity of p.}

5.3 Summary

The intent of this paper is to point out the relationship between the theory-building approach to knowledge representation taken by AI researchers and the more generally used model-building approach. In particular, it shows how they can be reconciled in a multiple representation system. Meta-level architecture is a natural way to build such a system and opens the possibility of substantial meta-level control in carrying out each operation. Finally, because the system is its own model of its theory, it's completely modifiable. The architecture should be useful in facilitating further research in meta-control, theory formation, automatic selection of data representation, and compiling for a heterogeneous environment of processors.

References

Balzer, R. Automatic Programming

Bobrow, D. & Winograd, T. KRL paper

Brachman, R. What's in a Concept

Brachman, R. Epistemological Foundations of Semantic Networks

Brachman, R. default paper

deKleer, J., etc. AMORD

Davis, R. & Buchanan, B. G. Meta-Rules

Doyle, J. A Model for Deliberation, Action, and Introspection, TR-581, M. I. T. Artificial Intelligence Laboratory, May 1980.

Fahlman, S. E. The Intersection Problem

Floyd, R. program verification paper

Friedland, P. & Smith, R. Units manual

Fox, M. inheritance paper

Genesereth, M. R. Canonicity in Rule Systems, Proceedings of the Symposium on Symbolic and Algebraic Manipulation, Springer-Verlag, June 1979.

Genesereth, M. R. Metaphors and Models, Proceedings of the First National Conference on Artificial Intelligence, Aug. 1980.

Genesereth, M. R., Greiner, R., Smith, D. E. MRS Manual

Greiner, R. D. & Lenat D. B. A Representation Language Language, Proceedings of the First National Conference on Artificial Intelligence, Aug. 1980.

Hayes, P. H. Logic of Actions

Hendrix, G. Partitioned Semantic Nets

Hewitt, C. thesis

Hewitt, C. intentions

Lenat, D. B., Hayes-Roth, F., Klahr, P. Cognitive Economy

McCarthy, J. Advice Taker

McCarthy, J. & Hayes, P. H. problems paper

McDermott, D. & Doyle, J. Non-monotonic Logic

McDermott, D. & Sussman, G. Conniver Manual

Minsky, M. Frames

Moses, J. simp paper

Mylopoulos procedural semantics

Schubert, L.

Sloman, A.

Smith, D. E. CORLL Manual

Sridharan, etc. AIMDS

Sussman, G. J. Microplanner Reference Manual

Warren Prolog

Weyhrauch, R. W. Prolegomena to a Theory of Formal Reasoning, AI Journal.

Winograd, T.

Woods, W. A. What's in a Link?

QA-3, QA-4, Simula, FRL


FILM
2-83